

# Par4All developer guide

## HPC Project

Mehdi AMINI    Amaury DARSCH    Onil GOUBIER    Serge GUELTON  
Ronan KERYELL    Janice ONANIAN-MCMAHON    Grégoire PÉAN  
Claire SEGUIN    Mickaël THIEVENT    Pierre VILLALON

April 23, 2012

**This manual is for Par4All version 1.4**

This document can be found in PDF format on [http://download.par4all.org/doc/developer\\_guide/par4all\\_developer\\_guide.pdf](http://download.par4all.org/doc/developer_guide/par4all_developer_guide.pdf) and in HTML on [http://download.par4all.org/doc/developer\\_guide/par4all\\_developer\\_guide.htdoc](http://download.par4all.org/doc/developer_guide/par4all_developer_guide.htdoc).

## 1 Introduction

PAR4ALL is a platform that merges various open source developments to ease the migration of sequential software to multicore and other parallel processors.

PAR4ALL is mainly developed by HPC Project, MINES ParisTech/CRI, Institut Télécom/Télécom Bretagne and others.

This document describes the internal organization of PAR4ALL and how its construction relies on GIT repositories, SVN repositories and other projects.

This document describes also the internal workings of PAR4ALL. This information may be useful not only for PAR4ALL core developers but also for advanced users desiring more functionality from PAR4ALL.

Since PAR4ALL relies on other tool projects, the documentation of these other projects should also be consulted independently and is not included here.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Advanced installation</b>	<b>3</b>
2.1	The p4a_setup.py compilation and installation script . . . . .	4
2.1.1	Options . . . . .	4
2.1.2	Setup Options . . . . .	4
2.1.3	General options . . . . .	5
<b>3</b>	<b>Committing using git</b>	<b>6</b>
<b>4</b>	<b>Collaborative repositories</b>	<b>6</b>
4.1	Public repositories . . . . .	6
4.2	Private repositories . . . . .	7
4.3	Packages . . . . .	7
4.4	Directory organization . . . . .	7

<b>5</b>	<b>Repositories and work-flow</b>	<b>8</b>
5.1	The p4a_git script to deal with workflow . . . . .	9
5.1.1	p4a_git and git typical use cases . . . . .	9
5.1.2	More advanced use cases for p4a_git . . . . .	10
5.1.3	p4a_git options . . . . .	11
5.2	PolyLib work-flow . . . . .	12
5.3	PIPS work-flow . . . . .	13
5.3.1	Compilation from sources outside of Par4All . . . . .	13
5.3.2	Debugging PIPS and p4a . . . . .	13
5.3.3	Internal organization . . . . .	14
5.4	PIPS-GFC extension workflow . . . . .	15
5.4.1	New version . . . . .	15
5.4.2	Old version (historical) . . . . .	15
<b>6</b>	<b>Infrastructure Setup and Maintenance</b>	<b>16</b>
6.1	Scripts for an everyday work . . . . .	16
6.2	Scripts for debugging . . . . .	16
6.3	Scripts used to setup the infrastructure . . . . .	17
6.4	Script to manage documentation . . . . .	17
6.5	Creating distributions . . . . .	17
6.5.1	Options . . . . .	17
6.5.2	Packing Options . . . . .	18
6.5.3	General options . . . . .	18
6.6	Making releases . . . . .	19
6.6.1	Options . . . . .	19
6.6.2	Coffee Options . . . . .	19
6.6.3	Setup Options . . . . .	19
6.6.4	Packing Options . . . . .	21
6.6.5	General options . . . . .	21
<b>7</b>	<b>p4a architecture</b>	<b>22</b>
<b>8</b>	<b>Examples and demos</b>	<b>23</b>
<b>9</b>	<b>Validation</b>	<b>23</b>
9.1	p4a_validate utility . . . . .	23
9.1.1	Example . . . . .	23
9.1.2	Option list of p4a_validate . . . . .	24
9.2	p4a_validate.class.py validation script . . . . .	25
<b>10</b>	<b>Branches</b>	<b>25</b>
<b>11</b>	<b>Useful git tricks for Par4All</b>	<b>26</b>
11.1	Files committed to the wrong branch . . . . .	26
11.2	Using GIT to push some branch on a test machine to try a PAR4ALL version . . .	27
11.3	The history has been rewritten on the server or how to resolve an uchrony with GIT	28
<b>A</b>	<b>Various script details</b>	<b>28</b>
A.1	The p4a_recover_includes script . . . . .	29
A.1.1	Options . . . . .	29
A.1.2	Debug options . . . . .	29
A.2	The p4a_post_processor.py script from Par4All Accel . . . . .	29
A.2.1	Options . . . . .	29
A.2.2	Debug options . . . . .	30
A.3	The optparse_help_to_tex help documentation to T <sub>E</sub> X compiler . . . . .	30

A.3.1 Options . . . . .	30
<b>B Tools, tips and tricks for developers</b>	<b>30</b>

## 2 Advanced installation

For a more simple installation please refer to [http://download.par4all.org/doc/installation\\_guide](http://download.par4all.org/doc/installation_guide).

The installation process can be modified by passing options to `p4a_setup.py`. For example, to skip the (re)compilation of the package `polylib`, `--skip-polylib` is used. The complete set of options is described in § 2.1.

The `make` command has options for speeding up the compilation. For example, to run on 8 processes, the `--jobs=8` option is added.

To compile PAR4ALL directly from source packages that are not inside the PAR4ALL directory hierarchy, see § 4.4. In this case, `--PACKAGE-src=...` specifies the location of the sources of *package*, (e.g., `--pips-src=...` or `--polylib-src=...`). If all these options are set (e.g., if the source locations point to working copies of the PIPS SVN), then the `p4a-own` branch can be used for compilation. For the PIPS packages, there must be links to `nlpmake/makes` to enable compilation. See § 5.3.1 for more information on alternate source locations.

The options passed to `configure` for a package can be changed by using the `--PACKAGE-conf-options=...`

For example, to compile PIPS with Fortran 95 support, use:

```
p4a_setup.py --only=pips --pips-conf-options="--enable-tpips --enable-pyips \
--enable-hpfc --enable-fortran95" --reconf --no-final
```

To recompile and intall PIPS after subsequent modifications, use:

```
p4a_setup.py --only=pips --no-final
```

Beware that if `--pips-src` is used to designate a PIPS directory that has been previously build in a classical way (such as the classical SVN build), the compilation will fail because of incorrect dependencies between the files contained therein (which are not related to PAR4ALL build directory location). Prior to running the script, run the following in the PIPS source directory.

```
make clean
```

To recompile a part of PAR4ALL (for example to debug one of the components) without using `p4a_setup.py`, change to the directory `$P4A_ROOT/build` in the relevant component. For example, change to `$P4A_ROOT/build/newgen` and type:

```
make
```

and upon completion, type:

```
make install
```

to complete the build process. The direct `install` approach can be used to avoid testing the compilation *a priori*. However, the 2-step approach is most useful if, for example, there is a debug session or a validation on the installed version and at the same time, a parallel development and verification effort. One can work in the latter without invalidating the running version.

The classical AUTOTOOLS environment variables can be used to influence the compilation, however, some are directly set up from the `p4a_setup.py` script. Therefore, to set the `CFLAGS` and `CPPFLAGS` to compile PAR4ALL in debug mode, value changes should be passed through a `p4a_setup.py` option. For example,

```
--configure-options="CFLAGS='-ggdb -g3 -Wall -std=c99'"
```

sets debugging options that allow `gdb` to access macro definitions that are heavily used in PIPS. Indeed this case is so common when debugging PAR4ALL that the `--debug` or `-g` options are just doing this for simplicity.

## 2.1 The `p4a_setup.py` compilation and installation script

The compilation and installation of PAR4ALL is controlled by the `p4a_setup.py` script, with the usage and options described in this section.

Usage: `p4a_setup.py [options]`; run `p4a_setup.py --help` for options

### 2.1.1 Options

`-h, --help`: show this help message and exit

### 2.1.2 Setup Options

`-R, --rebuild`: Rebuild the packages completely.

`-C, --clean`: Wipe out the installation directory before proceeding. Implies `-R` and not skipping any package.

`--skip-polylib, --sp`: Skip building and installing of the polylib library.

`--skip-newgen, --sn`: Skip building and installing of the newgen library.

`--skip-linear, --sl`: Skip building and installing of the linear library.

`--skip-pips, --sP`: Skip building and installing of PIPS.

`--skip-examples, --sE`: Skip installing examples.

`-s PACKAGE, --skip=PACKAGE`: Alias for being able to say `-s pips` (besides `--skip pips`), for example. `-sall` or `--skip all` are also available and means 'skip all', in which case only final installation stages will be performed.

`-o PACKAGE, --only=PACKAGE`: Build only the selected package. Overrides any other option.

`-r PACKAGE, --reconf=PACKAGE`: Always run autoreconf and configure selected packages. By default, only packages which lack a Makefile will be reconfigured. If `--rebuild` is specified, all packages will be reconfigured.

`--root=DIR`: Specify the directory for the Par4All source tree. The default is to use the source tree from which this script comes.

`-P DIR, --packages-dir=DIR, --package-dir=DIR`: Specify the packages location. By default it is `<root>/packages`.

`-p DIR, --prefix=DIR`: Specify the prefix used to configure the packages. Default is `/usr/local/par4all`.

`-b DIR, --build-dir=DIR`: Specify the build directory to be used relatively to the root directory as specify the `--root` option. Default to build

`--polylib-src=DIR`: Specify polylib source directory.

`--newgen-src=DIR`: Specify newgen source directory.

`--linear-src=DIR`: Specify linear source directory.

`--pips-src=DIR`: Specify PIPS source directory. When changing the directory, do not forget to reconfigure since this is when the source location is taken into account.

- `--nlpmake-src=DIR`: Specify nlpmake source directory.
- `-c OPTS, --configure-options=OPTS, --configure-flags=OPTS`: Specify global configure options. Default is `'--disable-static CFLAGS='-O2 -std=c99'` OR `'--disable-static CFLAGS='-ggdb -g3 -O0 -Wall -std=c99'` if `--debug` is specified.
- `-g, --debug`: Set debug CFLAGS in configure options (see `--configure-options`). Please note that this option has NO EFFECT if `--configure-options` is manually set.
- `--polylib-conf-options=OPTS, --polylib-conf-flags=OPTS`: Specify polylib configure opts (appended to `--configure-options`).
- `--newgen-conf-options=OPTS, --newgen-conf-flags=OPTS`: Specify newgen configure options (appended to `--configure-options`).
- `--linear-conf-options=OPTS, --linear-conf-flags=OPTS`: Specify linear configure options (appended to `--configure-options`).
- `--pips-conf-options=OPTS, --pips-conf-flags=OPTS`: Specify PIPS configure options (appended to `--configure-options`). Defaults to `--enable-tpips --enable-pyps --enable-hpfc --enable-fortran95`. Setting this option will reset the default value. Note that several flags can be set like this : `--pips-conf-options "--enable-tpips --enable-pyps --enable-doc"`
- `-m OPTS, --make-options=OPTS, --make-flags=OPTS`: Specify global make options.
- `--polylib-make-options=OPTS, --polylib-make-flags=OPTS`: Specify polylib make opts (appended to `--make-options`).
- `--newgen-make-options=OPTS, --newgen-make-flags=OPTS`: Specify newgen make options (appended to `--make-options`).
- `--linear-make-options=OPTS, --linear-make-flags=OPTS`: Specify linear make options (appended to `--make-options`).
- `--pips-make-options=OPTS, --pips-make-flags=OPTS`: Specify PIPS make options (appended to `--make-options`).
- `-j COUNT, --jobs=COUNT`: Make packages concurrently using COUNT jobs.
- `-I, --no-install`: Do not install any package (do not run make install for any package). NB: this might break the compilation of packages depending on the binaries of uninstalled previous packages.
- `-F, --no-final`: Skip final installations steps in install directory (installation of various files). NB: never running the final installation step will not give you a functional Par4All build.

### 2.1.3 General options

- `-v, --verbose`: Run in verbose mode: each `-v` increases verbosity mode and display more information, `-vvv` will display most information.
- `--log`: Enable logging in current directory.
- `--report=YOUR-EMAIL-ADDRESS`: Send a report email to the Par4All support email address in case of error. This implies `--log` (it will log to a distinct file every time). The report will contain the full log for the failed command, as well as the runtime environment of the script like arguments and environment variables.

- report-files:** If `--report` is specified, and if there were files specified as arguments to the script, they will be attached to the generated report email. **WARNING:** This might be a privacy/legal concern for your organization, so please check twice you are allowed and willing to do so. The Par4All team cannot be held responsible for a misuse/unintended specification of the `--report-files` option.
- report-dont-send:** If `--report` is specified, generate an `.eml` file with the email which would have been send to the Par4All team, but do not actually send it.
- z, --plain, --no-color, --no-fancy:** Disable coloring of terminal output and disable all fancy tickers and spinners and this kind of eye-candy things :-)
- no-spawn:** Do not spawn a child process to run processing (this child process is normally used to post-process the PIPS output and reporting simpler error message for example).
- execute=PYTHON-CODE:** Execute the given Python code in order to change the behaviour of this script. It is useful to extend dynamically Par4All. The execution is done at the end of the common option processing
- V, --script-version, --version:** Display script version and exit.

### 3 Committing using git

In general, to commit changes to the GIT repository, the `p4a-own` branch should be used and not any of the PAR4ALL sub-packages. If a branch must be used, see section 4.3 to ensure consistency with the PAR4ALL compilation process. It is highly advisable to compile work before committing it on the central repositories and to make sure that all files are committed so that the code compiles in all user environments. In addition to providing consistency, this has the added advantage of allowing one to test one's code before releasing it for general consumption! ☺

A nice feature of GIT over SVN is that since the commit is separated from the publication, a commtted state can be tested independently from the rest of the team before pushing to the global server.

For example, one can create a light<sup>1</sup> clone with

```
git clone --branch p4a par4all par4all-compile
```

and after testing and committing modifications inside the `par4all` working copy, one does the same into the `par4all-compile` working copy after a `git pull`. If some files are lacking from the commit, GIT will be detect the discrepancy.

Afterwards, a `git push` into the central PAR4ALL repository will have fewer associated risks.

## 4 Collaborative repositories

### 4.1 Public repositories

There are several GIT repositories used by the project.

To have access without authentication and only for reading/cloning, use the `git:` prefix instead of `ssh:`, e.g., `git://git.hpc-project.com/git/par4all.git`.

The main repository for the project is located at `ssh://git.hpc-project.com/git/par4all.git`

This repository can be viewed with a WWW browser at `https://git.hpc-project.com/cgit/par4all`

To get directly involved into the project with full commit capability directly into the repositories, ask HPC Project.

---

<sup>1</sup>Because the objects are shared with symbolic links and not copied, since we did not use the `file://` syntax.

There are also ancillary GIT repositories that provide a GIT interface to the trunk of the SVN repositories for the PIPS components from CRI:

- `ssh://git.hpc-project.com/git/svn-linear.git`
- `ssh://git.hpc-project.com/git/svn-newgen.git`
- `ssh://git.hpc-project.com/git/svn-nlpmake.git`
- `ssh://git.hpc-project.com/git/svn-pips.git`
- `ssh://git.hpc-project.com/git/svn-validation.git`

These ancillary gateways only include the `trunk` history since the CRI branches are not public.

In the case of `nlpmake`, another GIT SVN gateway at the top level has been used for the integration with `trunk`, `branch` and `tag`. This is due to the fact that `nlpmake` started without the standard layout, which was added later at approximately revision 750. Because of this prior history, this gateway is not published in a public git.

## 4.2 Private repositories

There is a private directory shared between core developers and used mainly for validation of the project on non public codes, benchmarks, demos, and for developing private reports, phases, scripts and so on: `ssh://git.hpc-project.com/git/par4all-private.git`

For HPC Project-confidential information, `ssh://git.hpc-project.com/git/par4all-private-hpc.git` is used.

Other repositories can be created and used on demand according to the needs of evolving private collaborations.

## 4.3 Packages

PAR4ALL integrates different tools from different projects. Currently, PAR4ALL is composed of PIPS, PIPS-GFC, POLYLIB, with some extensions. Each project is included in PAR4ALL as a package and is placed in a directory inside the `package` top-level directory. These directories exist as GIT subtrees to ease revision control and integration, even when not connected to the SVN server.

Since PAR4ALL is an integration project, to modify or develop in a particular package, please work in the upstream package and not in PAR4ALL<sup>2</sup>. Working in this manner facilitates compilation of PAR4ALL with package sources outside of PAR4ALL (see § 2.1), which can be committed into their own upstream version control systems.

## 4.4 Directory organization

The PAR4ALL distribution contains the following directories:

`build` is created when compiling the various PAR4ALL packages from the `AUTOTOOLS`;

`doc` contains the sources of the PAR4ALL documentation, including those for the user and the programmer as well as those about the infrastructure:

`organization` stores indeed the sources of this document;

`p4a_coding_rules` contains some coding rules applications should respect to be dealt seamlessly by PAR4ALL;

---

<sup>2</sup>In fact, this method is used for PAR4ALL development at Rensselaer Polytechnic Institute; global developments in PIPS are created directly and easily into `package/PIPS`. However, the re-integration becomes more difficult since it requires extracting the patch history in `package/PIPS` and merging it into the PIPS upstream SVN repository.

`simple_tools/p4a_article` describes the PAR4ALL capabilities for the end user and is the user manual;

`p4a_slides` is a very simplified version of the previous manual as slides;

`examples` contains some examples and benchmarks to exercise PAR4ALL. These examples also encompass PAR4ALL demos;

`packages` contains the different components of PAR4ALL:

- PIPS contains the components of PIPS framework itself, including:
  - `linear`: the main linear library of PIPS;
  - `newgen`: the object management infrastructure used by PIPS;
  - `nlpmake`: the makefile common infrastructure used by all the PIPS components
  - `pips`: the PIPS core;
  - `validation`: the validation of PIPS;
- `pips-gfc` contains a GCC 4.4 source patch to be compiled and linked with PIPS that adds a Fortran 95+ parser to PIPS;
- `polylib` contains the POLYLIB linear library source;

`src` contains sources of tools used for the internal organization of PAR4ALL itself, such as repository and product management, product publication, or run-time management (e.g., `p4a_accel`);

*PREFIX-DIR* contains the usable PAR4ALL is installed after compilation:

- `bin` contains the executable programs from PAR4ALL;
- `doc` contains the generated documentation for the PAR4ALL infrastructure;
- `etc` contains some generated configuration files;
- `examples` contains with some examples to exercise PAR4ALL;
- `include` contains the include files used for the compilation of PAR4ALL;
- `lib` contains the libraries used to run PAR4ALL;
- `makes` contains `make`-file tools for the PIPS validation;
- `RELEASE-NOTES.txt` is the release notes of this PAR4ALL instance;
- `share` contains shared files for run-time and configuration files
- `VERSION` is the current version of this PAR4ALL instance.

## 5 Repositories and work-flow

For history tracking and collaborative development, PAR4ALL relies on a main GIT repository that is accessed by PAR4ALL developers, users, integrators, and the production and quality assurance team.

The script `p4a_git` is used to automate the management of the workflow.

Since PAR4ALL extends some tools such as PIPS, there are some ancillary repositories to ease the impedance matching between PAR4ALL and those other projects.

Since POLYLIB is already a GIT repository, it is simply fetched as a remote GIT into the PAR4ALL at the right place.

The process is more complex with PIPS, which is a project split in 5 independent SVN repositories requiring more effort for coherent presentation. Toward this end, SVN-GIT gateways are used to offer freedom of development and independence from other review hierarchies. Using these gateways, light branches are stored into a common GIT view of the PIPS SVN repositories; these light branches can be pushed back into the PIPS SVN.

## 5.1 The p4a\_git script to deal with workflow

The `p4a_git` script is used to manage the workflow of PAR4ALL involving all the related GIT and SVN repositories.

For easy rollback to a robust state, a branch can be created with `git checkout -b` and tested independently prior to merging with the public repository. If the independent branch fails, then it can be deleted without impacting other work. See the `git reset` documentation for more details.

Some use cases of `p4a_git` require that the `P4A_TOP` environment variable be set to the top directory of the PAR4ALL infrastructure wherein reside the various GIT working copies involved in the project.

The following section presents some examples, after which the full list of options is described in detail.

### 5.1.1 p4a\_git and git typical use cases

The PAR4ALL `p4a` reference branches are used to construct PAR4ALL and potentially contain integration work performed by PAR4ALL team members beyond the current release. To retrieve an up-to-date version of a reference branch, use:

```
p4a_git --branch-action-name p4a "git checkout \${branch}; git pull origin \${branch}"
```

Work performed on PAR4ALL itself without integrating new versions of upstream packages is first integrated into the `p4a-own` branch using: `git stuff`:

```
git checkout p4a
git merge p4a-own
```

When the results are validated and reread (for example verify with `gitk` that something has not been committed by error on `p4a` branch), work is published to the main `p4a` branch using:

```
git push origin p4a-own
git push origin p4a
```

A typical example to build a PAR4ALL with the latest PIPS version into the reference `p4a` branch infrastructure (after having merged reference branches as explained previously) is as follows:

```
p4a_git --build-new-p4a
```

Indeed, the effect is to chain the following actions that can be manually used to tweak the aggregation process (for example to skip a wrong or out-of-phase commit in PIPS):

- fetch the most recent developments of upstream packets (PIPS, POLYLIB...) into the `git svn` repositories in the `CRI-git-svn` folder:

```
p4a_git --update-git-svn
```

- merge the new versions of the upstream packages into the PAR4ALL reference branches, such as `CRI-pips`:

```
p4a_git --fetch-remote-git
```

- pull all the component branches (such as `CRI-pips`) into the PAR4ALL integration branch hierarchy (such as `p4a-pips`):

```
p4a_git --pull-remote-git
```

This is where you can play around upstream flows. For example you can afterwards send the `p4a-pips` back a few commits, cherry-pick from `CRI-pips` some useful contributions among broken commits.

Note that the 2 previous instructions work into `$P4A_ROOT`, so make sure it does not point to a default installation directory such as `/usr/local/par4all`. You can update `$P4A_ROOT` or use the `--root .` option if you worked in the `par4all` working copy;

- fuse all the component branches into the `PAR4ALL` integration branch hierarchy:

```
p4a_git --aggregate-branches
```

At the end of this sequence of steps, the most recent developments in all parts of `PAR4ALL` is contained in the `p4a` branch infrastructure. After testing and validation, this branch can be published<sup>3</sup> as the latest official version with:

```
p4a_git --branch-action git push origin \${branch}
```

Note that this may be quite verbose, so you may be more comfortable with a quite:

```
p4a_git --branch-action git push origin \${branch} >& /dev/null
```

If someone else updates a branch in the meantime, the new branches must be merged back into the local versions by re-executing the sequence of steps at the beginning of this section.

### 5.1.2 More advanced use cases for `p4a_git`

The previous section provides enough guidance for simple integration and maintenance of `PAR4ALL`. In this section, more advanced use cases are presented.

For example, to prepare the release candidate of `PAR4ALL 1.3`, a developer can create from `p4a` a new branch infrastructure named `p4a-1.3-rc`, with `p4a-1.3-rc-pips` for `PIPS`, `p4a-1.3-rc-polylib` for `POLYLIB`, etc., with each one starting from its relative `p4a`-prefixed branch infrastructure. To do this, use;

```
p4a_git --aggregate-branches p4a-1.3-rc p4a
```

To start the `p4a-0.3-alpha` branch infrastructure from `p4a-0.2-rc` branch infrastructure, use:

```
p4a_git --aggregate-branches p4a-0.3-alpha p4a-0.2-rc
```

Once the new branch infrastructure is created, you can use it, modify its component branches and apply the merge integration work-flow with for example:

```
p4a_git --aggregate-branches p4a-1.3-rc
```

to update the `p4a-1.3-rc` branch infrastructure.

When finished, these branches can be pushed for sharing or merged into the main `PAR4ALL` GIT repository `p4a` with:

```
p4a_git --branch-action-name p4a-very-personal git push origin \${branch}
```

To pull all the different `PAR4ALL` branches that may have been changed by others:

```
p4a_git --branch-action-name p4a "git checkout p4a\${suffix}; git pull origin p4a\${suffix}"
```

or in a more compact way:

```
p4a_git --branch-action "git checkout \${branch} git pull origin \${branch}"
```

---

<sup>3</sup>But before this, think to verify it is correct with a tool like `gitk` because it may be cumbersome for the community to remove some wrongly pushed commits...

To set up all the tracking branches for version p4a-0.2-alpha before using them:

```
p4a_git --branch-action-name p4a-0.2-alpha git branch \
remotes/origin/\$branch \
```

Developing and trying a new version of the scripts dealing with the PAR4ALL infrastructure itself, such as `p4a_setup.py`, is complicated by the fact that script execution changes the branch. This may cause the script to disappear during execution since it has not been committed into the final branch<sup>4</sup>.

To test script modifications, first clone the repository into a new one in the `$P4A_TOP` directory. Then, set `P4A_ROOT` to the new repository. For a more independent development environment, one could also set `P4A_TOP` to a new world.

If you are developing the PAR4ALL infrastructure itself, since `p4a_setup.py` install PAR4ALL and rebuild a new configuration file at the end, to test the new `p4a_git` script for example for the construction of the new PAR4ALL, it is useful to set `P4A_ETC` to point to the directory that contains the `p4a_git_lib.bash` that will be used for example, to speed up the bootstrapping process. Subsequent executions of the `p4a_setup.py` under revision will act on the new sand-box repository working copy.

To test a `par4all` repository before the final push by merging and building into a new `p5` repository, use the following sequence of steps:

```
cd $P4A_TOP
# Create a new clone from par4all in the same directory. Note that git
# objects are hard-linked to the first one for space efficiency:
git clone par4all p5
# Where we will work:
export P4A_ROOT=$P4A_TOP/p5
cd $P4A_ROOT
# Select the branch version you want to try, normally p4a, but can be
# something more specific, such as p4a-1.3-rc:
git checkout p4a
# Indented part is not useful for a normal developer:
# If you are working on the Par4All infrastructure itself and its tools.
export P4A_ETC=$P4A_TOP/par4all/src/dev
# Optionally Add remotes to the PIPS git-svn & PolyLib git gateway if we want to
# test the production of a new version:
$P4A_TOP/par4all/src/dev/p4a_git --add-remotes
# Here you may update the svn-git gateways and get all the sources at
# the right place with a brand new version:
$P4A_TOP/par4all/src/dev/p4a_git --build-new-p4a
# For a normal developer, simply build Par4All
$P4A_ROOT/src/dev/p4a_setup.py <your options>
# Source the new environment. Be carefull, it overrides the P4A_ETC
# variable above:
source $P4A_ROOT/run/etc/par4all-rc.sh
# Just try it now...
```

### 5.1.3 p4a\_git options

The `p4a_git` options are available in short and long forms:

`-h` or `--help` display the usual help message;

`-v` or `--verbose` increase the verbosity of the script. For example, if used twice, the script enters into command tracing mode;

---

<sup>4</sup>This occurs when the script modifies its own branch. ☺

- n or --build-new-p4a chain most following phases to have a new PAR4ALL source version;
- u or --update-git-svn update the PIPS GIT-SVN gateways that are into \$P4A\_TOP/CRI-git-svn;
- g or --recursive-git-svn apply a GIT command to all the git working copies inside the current directory recursively. If no argument is given, a `git svn rebase` is done;
- f or --fetch-remote-git fetch the objects from the remote GIT repositories (the PIPS GIT-SVN gateways and the POLYLIB GIT);
- p or --pull-remote-git pull the objects from the remote GIT repositories into their respective branches and update the p4a branch hierarchy to point to the last version of PAR4ALL;
- m or --aggregate-branches [*<to-prefix>* [*<origin-prefix>*]] merge all the p4a-like branch infrastructure into the *<to-prefix>* branch architecture. If *<to-prefix>* is null, work into p4a default branch infrastructure. If *<origin-prefix>* is null, use the same as *<to-prefix>*, that means we aggregate the *<to-prefix>* branch infrastructure without using external branches. If the *<to-prefix>* branch infrastructure does not exist, it is created from the *<origin-prefix>* branch infrastructure;
- b or --branch-action-name *<branch-prefix>* *args+* apply a shell-script to a branch hierarchy with name starting with *<branch-prefix>*. The branch name is available in the `$branch` variable and the suffix available in the `$suffix` variable. Do not forget to escape special shell characters in the final shell;
- a or --branch-action *args+* same as --branch-action-name p4a ... to deal with default p4a branch hierarchy;
- add-remotes create the remotes pointing to \$P4A\_CRI\_GIT\_SVN and on the POLYLIB GIT;
- r or --root *<directory>* to change the git working repository.

## 5.2 PolyLib work-flow

The basic workflow for POLYLIB is first to develop new features into the original POLYLIB GIT repository, and then to fetch into the PAR4ALL GIT repository and select particular features for the `packages/polylib` of PAR4ALL.

The script `p4a_git` is used to automate the management of the workflow.

`p4a_git --pull-remote-git` pulls the POLYLIB into the `polylib` branch so that global modifications can be applied there if needed and later `git merged` into the local working branch. In this way, global modifications are persistent and available to everybody.

The following presents information on the direct management of the POLYLIB workflow.

Since the POLYLIB is already in a GIT repository, the `polylib` is simply a remote reference in the PAR4ALL GIT. Therefore, to import the latest POLYLIB development into the PAR4ALL GIT for inspection and inclusion, fetch POLYLIB with:

```
git fetch ICPS/polylib
```

then merge the desired feature with

```
git merge -s subtree remotes/polylib/master
```

or using any tree identifier. The `-s subtree` is necessary since in PAR4ALL the POLYLIB files are not at the top-level directory. This should be done into the `polylib` branch for compatibility with the workflow chosen in PAR4ALL.

## 5.3 PIPS work-flow

### 5.3.1 Compilation from sources outside of Par4All

When developing PAR4ALL packages, it is of critical importance to test them *before* including them into PAR4ALL. For example, a new PIPS version is developed in the CRI SVN repository before building PAR4ALL to test the integration.

This process has been described in the compilation manual; further details are provided in the following example.

Assuming there is a SVN working copy of PIPS (including the `trunk` into `prod`, your branch in `pips_dev`, etc.; refer to the PIPS developer guide for more information), use the `--pips-src=` to make PAR4ALL point to this directory.

Verify that the correct `PAR4ALL` variable is set in your shell, that the previous configuration file from the PIPS SVN environment from CRI is not automatically sourced in your shell and that there are no incorrect environment variables from previous invocations of a script (e.g., `PATH`, etc.)

Clean this directory to remove any dangling dependencies that may result from previous usage by PAR4ALL of a PIPS directory that was installed in the classical manner. This is accomplished using:

```
cd <where the wanted PIPS is>
make clean
```

Subsequently, a new PAR4ALL version is built using the new PIPS version in debug mode using the following command, which uses four compilation processes (assuming a 4-core machine):

```
src/simple_tools/p4a_setup.py -vvv --pips-src=<where the desired PIPS is> --debug --jobs=4
```

After building the complete infrastructure, subsequent changes to PIPS do not require rebuilding all packages or reconfiguring PIPS. The following command recompiles PIPS only and skips the installation of other PAR4ALL components:

```
src/simple_tools/p4a_setup.py -vvv --pips-src=<where the wanted PIPS is> \
--debug --only=pips --no-final --jobs=4
```

An example script performs all of the above actions can be found in:  
`src/simple_tools/p4a_setup_with_my_PIPS:`

```
1  #! /bin/sh
   # To compile p4a with my own copy of PIPS, in debug mode, with 4 processes
   # and in verbose mode.
6  # Can take other arguments from the command line.
   # Guess that you have a copy of the svn trunks in
   # ../../PIPS/git-svn-work and the Polylib in ../polylib
   # --pips-conf-opts=--enable-fortran95
11 src/simple_tools/p4a_setup.py --polylib-src ../polylib \
   --nlpmake-src ../../PIPS/git-svn-work/nlpmake \
   --newgen-src ../../PIPS/git-svn-work/newgen \
   --linear-src ../../PIPS/git-svn-work/linear \
   --pips-src ../../PIPS/git-svn-work/pips \
16 -g -j4 -vvv -z $*
```

### 5.3.2 Debugging PIPS and p4a

To debug `tpips` in PAR4ALL, it is sufficient to run `gdb tpips5` and then run `the.tpips` file in parameter.

PyPS programs are Python programs that use PIPS libraries. To debug PyPS programs, launch `gdb python` and inside the debugger and run the program with `run the.PyPS.file.py`

---

<sup>5</sup>Or an equivalent debugger or frontend such as Eclipse or Emacs. For example with Emacs, use `M-x gdb` and then select many windows in the `Gud/Gdb-UI` menu.

`p4a` is also a Python program, however, it requires several arguments and launches many processes. This complicates the debugging of PIPS directly.

One way to debug PIPS is to ask the debugger to follow the `fork()` and `exec()` system calls. This can be achieved by setting `detach-on-fork` off in `gdb` and also looking at `follow-exec-mode`.

A second easier way is to prevent `p4a` from launching processes. This is accomplished with the `--no-spawn` option. Avoiding colors and other decoration by using the `--plain` is also useful when debugging.

The preferred method for debugging `p4a` from `gdb` is to launch first `gdb python` or with Emacs and then run `p4a` with:

```
run /usr/local/par4all/bin/p4a --plain --no-spawn
-o hyantes-static-99_openmp hyantes-static-99.c -lm
```

To run your own version of `p4a`, add the directory containing your `p4a .py` libraries to the `PYTHONPATH` environment variable.

For memory debugging with Valgrind, see section 6.2.

### 5.3.3 Internal organization

The work-flow related to PIPS is complicated by the need to consolidate data from 5 different SVN repositories. The script `p4a_git` is used to automate the management of this workflow. Further details are provided in this section.

The basic PIPS work-flow is to develop into the 5 original PIPS SVN repositories at MINES ParisTech/CRI and to import the selected developments into the PAR4ALL GIT.

The infrastructure that supports this work-flow consists of 5 GIT repositories that are gateways to the original SVN repositories. To avoid some naming complexity, these gateways exist only on the laptop of Ronan KERYELL and are used as remotes into the PAR4ALL GIT. To synchronize these gateways to the latest version of the PIPS SVN repositories, run `pips_git` in the directory owning these GIT-SVN repositories.

Since the gateway provides a GIT interface to the original PIPS SVN repositories and GIT is more powerful than SVN, some users may want to develop code into PIPS using GIT via the gateways. For example, development and verification can proceed with with common branches in GIT before pushing into the PIPS SVN trunk. To provide a public interface for programmers, the GIT gateways are pushed to a public GIT repository. These public gateways are synchronized regularly and manually with the PIPS SVN. However, SVN may not be credited with the right owner of the commit but with the one running the gateway, which is not accurate. Therefore, the preferred method is to develop code directly in the original PIPSSVN repository.

The 5 public gateway GIT repositories are also defined as 5 remotes into the PAR4ALL GIT:

```
remotes/CRI/linear
remotes/CRI/newgen
remotes/CRI/nlpmake
remotes/CRI/pips
remotes/CRI/validation
```

These remote repositories are merged into PAR4ALL in the following respective subtree directories:

```
packages/PIPS/linear
packages/PIPS/newgen
packages/PIPS/nlpmake
```

packages/PIPS/pips

packages/PIPS/validation

To import the latest PIPS development into the PAR4ALL GIT for inspection and to choose these projects for inclusion, fetch the desired repositories with:

```
git fetch CRI/linear
git fetch CRI/newgen
git fetch CRI/nlpmake
git fetch CRI/pips
git fetch CRI/validation
```

This can also be achieved with `p4a_fetch_all`, which includes the POLYLIB part.

After the import, merge the desired feature with a `git merge -s subtree` from `remotes/CRI/.../master` with:

```
git merge -strategy=subtree remotes/CRI/linear/master
git merge -strategy=subtree remotes/CRI/newgen/master
git merge -strategy=subtree remotes/CRI/nlpmake/master
git merge -strategy=subtree remotes/CRI/pips/master
git merge -strategy=subtree remotes/CRI/validation/master
```

or from any tree identifier to do more precise version selection. The `-s subtree` is necessary since in PAR4ALL the PIPS files should not be at the top-level directory.

To pull everything at once for testing, use the `p4a_git` script described in § 5.1. Remember that it is preferable to create a branch with a `git checkout -b` before pulling everything, since the branch can be deleted after testing for easy rollback.

## 5.4 PIPS-GFC extension workflow

### 5.4.1 New version

In the work of Mehdi AMINI, the branch `p4a-gcc-gfc-4.4.3` contains the sources of GCC-GFC 4.4.3 which includes Fortran 95. The branch is also named `p4a-gcc-gfc`, which is merged into `p4a-packages`.

During PAR4ALL construction, these sources are patched and compiled into the PIPS Fortran 95 parser.

For example, to add the new 4.4.4 version, use:

```
git checkout p4a-gcc-gfc
git checkout -b p4a-gcc-gfc-4.4.4
<do your work and commit>
git checkout p4a-gcc-gfc
git merge p4a-gcc-gfc-4.4.4
```

### 5.4.2 Old version (historical)

This section describes the past organization that was instituted at the beginning of the project and provides the historical context for the current organization.

This part is derived from Raphaël ROOSZIS into the `package/pips-gfc` directory. The branch `gcc-4.4.1` contains plain GCC core and Fortran 4.4.1 distribution.

The development of PIPS-GFC should be done in the branch `pips-gfc-4.4.1`. This branch should be merged with the branch `gcc-4.4.1` into a branch `pips-gfc+gcc-4.4.1` with a more global name `pips-gfc+gcc`. This is the one to be merged into the global `master` branch.

The branch `pips-gfc-4.4.1` should contain only files that differ from the GCC distribution. In this way, if we want to have more subtle construction methods later, it will be clearer how to get the real content.

The same branch structure exists for version 4.4.2.

To develop and test the PIPS-GFC extension, change to `pips-gfc-4.4.1` with

```
git checkout pips-gfc-4.4.1
```

and develop your code in this branch.

To test code, commit and change to the `pips-gfc+gcc-4.4.1` branch with

```
git checkout pips-gfc+gcc-4.4.1
```

and then merge with:

```
git merge pips-gfc-4.4.1
```

and compile. Upon completion, either commit or revert and then return to branch `pips-gfc-4.4.1`.

To avoid corrupting the branches `pips-gfc-4.4.1` and `pips-gfc+gcc-4.4.1`, it is preferable to create sub-branches and commit in the sub-branches. Work can then be merged back, after which the sub-branches can be deleted. The `--slashed` option can be used if you want to be modest about your gory hesitations ☺.

## 6 Infrastructure Setup and Maintenance

There are scripts to facilitate development.

### 6.1 Scripts for an everyday work

- to automate the PAR4ALL workflow and work with `git`, the `p4a_git` is available, as described in § 5.1;
- `p4a_post_processor.py` is used to generate programs that use the PAR4ALL Accel runtime from the PIPS output. More information on this script can be found in section A.2;
- `p4a_recover_includes` is used primarily to get standard `#include` back after PIPS digestion. See section A.1 for the use case;
- `p4a_setup.py` is used to compile and setup the entire PAR4ALL infrastructure and should be used at least for the first compilation. See § 2.1 for more details;
- `p4a_validate` is used to leverage the PIPS validation.

### 6.2 Scripts for debugging

- `p4a_recover_includes` can be used to evaluate the inclusion of PAR4ALL Accel to ease debugging this package without evaluating other preprocessor inclusions. See section A.1 for the use case;
- `p4a_valgrind` launches a command with Valgrind with a memory checker in paranoid mode, mainly with the options described in the PIPS development guide. As seen in section 5.3.2, to debug PIPS in `p4a`, it is useful to do the following:

```
p4a_valgrind python /usr/local/par4all/bin/p4a --plain --no-spawn
-o hyantes-static-99_openmp hyantes-static-99.c -lm
```

Of course you can also use `valgrind` instead of `p4a_valgrind` for less verbosity.

### 6.3 Scripts used to setup the infrastructure

The following scripts are used to bootstrap the PAR4ALL infrastructure and should only be used by advanced users in the case of dramatic failures. They are located in `src/dev` and should be used in the order presented as follows:

- `p4a_create_CRI_git_svn` is used once to create the PIPS SVN-GIT gateways. This script is included in the distribution to show the exact parameters used in case there is a future need to recreate the gateways.
- `p4a_import_external_gits` imports all the external GIT repositories into the PAR4ALL GIT repository. It should be used only once but is included as an example for other projects in case there is a future need to add other repositories.
- `p4a_apply_pips_patches` is used to patch the original PIPS files to fit the PAR4ALL architecture. It should be used only once, after external GIT import. N.B. this script is obsolete in the AUTOTOOLS version of PIPS;

### 6.4 Script to manage documentation

- `optparse_help_to_tex` is a small compiler that transforms the help output message of a command launched with `-h` (using the `optparse` format) into LaTeX code for inclusion in articles or slides. See § A.3 for more details;
- `p4a_doc` in `src/dev` is used to compile the PAR4ALL documentation, make a Doxygen version of PAR4ALL Accel run-time and publish it on the server. See `p4a_doc -h` for the instructions.

### 6.5 Creating distributions

The `p4a_pack.py` is used to build compressed tar balls or Debian `.deb` packet files.

The location of the PAR4ALL files is specified according to the `P4A_DIST` variable or `--dir` option. However, there may be hard-coded locations at configure and build time. Therefore, moving the installation directory afterwards and changing `P4A_ROOT` to follow the new location may be insufficient.

Because of this, the location should be chosen at build time. For example, to have a distribution built into `/usr/local/par4all`, first create a *writable* `/usr/local/par4all` (i.e., create this directory as super-user and change the owner using `chown`), and then run `p4a_setup.py`.

The directory is then packed into the desired format as in the following:

```
src/simple_tools/p4a_pack.py <--deb|--tgz> --revision=0.1-mybuild \  
--dir=/current/install/dir [--prefix=/final/install/dir]
```

If the default temporary directory does not have enough space to build up a new distribution (as our old machine used to build the 32-bit Debian distribution), you can change the `TMPDIR`<sup>6</sup> to a directory location when there is enough room (1+ GB).

Below are the usage and full option list for this command.

```
Usage: p4a_pack.py <--deb|--tgz|--stgz> [--version=0.1] [--append-date]  
[--dir=/current/install/dir] [--prefix=/final/install/dir] [--publish] [other  
options] [optional additional files to publish]; run p4a_pack.py --help for  
options
```

#### 6.5.1 Options

`-h, --help`: show this help message and exit

---

<sup>6</sup>Indeed, at the Python level, `TEMP` and `TMP` could be also used instead, but `dpkg-deb` is used to build the package and it only follows `TMPDIR`. So use `TMPDIR`...

### 6.5.2 Packing Options

- `--pack-dir=DIR`: Directory where the distribution to package is currently installed. Default is to take the root of the Git repository in which this script lies.
- `--deb`: Build a .deb package.
- `-T, --tgz`: Create a .tar.gz archive.
- `--stgz`: Create a source .tar.gz archive.
- `--arch=ARCH`: Specify the package architecture manually. By default, the current machine architecture is used.
- `--distro=DISTRO`: Specify the target distribution manually. By default, the current running Linux distribution is used.
- `--package-version=VERSION`: Specify version for packages.
- `--append-date, --date`: Automatically append current date & time to version string.
- `--publish`: Publish the produced packages on the server.
- `--publish-only=FILE`: Publish only a given file (.deb, tgz, stgz or even whatever for testing) without rebuilding it. Several files are allowed by using this option several times.
- `--retry-publish`: Retry to publish only files (.deb and/or tgz and/or stgz) from the local directory without rebuilding them. Be sure from what is in your local directory! To be used with p4a\_pack and not with p4a\_coffee
- `--release`: When publishing, put the packages in release directories instead of development ones.
- `--install-prefix=DIR`: Specify the installation prefix. Default is /usr/local/par4all.
- `-k, --keep-temp`: Do not remove temporary directories after script execution.
- `--pack-output-dir=DIR`: Directory where package files will be put (locally). Any existing package with the same name (exact same revision) will be overwritten without prompt. Defaults to current working directory: /home/keryell/projets/WildSystems/Par4All/par4all/doc/developer\_guide

### 6.5.3 General options

- `-v, --verbose`: Run in verbose mode: each -v increases verbosity mode and display more information, -vvv will display most information.
- `--log`: Enable logging in current directory.
- `--report=YOUR-EMAIL-ADDRESS`: Send a report email to the Par4All support email address in case of error. This implies --log (it will log to a distinct file every time). The report will contain the full log for the failed command, as well as the runtime environment of the script like arguments and environment variables.
- `--report-files`: If --report is specified, and if there were files specified as arguments to the script, they will be attached to the generated report email. WARNING: This might be a privacy/legal concern for your organization, so please check twice you are allowed and willing to do so. The Par4All team cannot be held responsible for a misuse/unintended specification of the --report-files option.
- `--report-dont-send`: If --report is specified, generate an .eml file with the email which would have been send to the Par4All team, but do not actually send it.

- z, --plain, --no-color, --no-fancy: Disable coloring of terminal output and disable all fancy tickers and spinners and this kind of eye-candy things :-)
- no-spawn: Do not spawn a child process to run processing (this child process is normally used to post-process the PIPS output and reporting simpler error message for example).
- execute=PYTHON-CODE: Execute the given Python code in order to change the behaviour of this script. It is useful to extend dynamically Par4All. The execution is done at the end of the common option processing
- V, --script-version, --version: Display script version and exit.

## 6.6 Making releases

After validating, a release is created by tagging the current branch at the current state.

There is also a `p4a_coffee.py` script to build and publish everything at once.

Usage: `p4a_coffee.py [options]`; run `p4a_coffee.py --help` for options

### 6.6.1 Options

- h, --help: show this help message and exit

### 6.6.2 Coffee Options

- here: Do not clone the repository, assume we are building from the Git tree where the script `../../src/simple_tools/p4a_coffee.py` lies.
- git-revision=VERSION: By default Par4All is built from the 'p4a' branch when cloning or the current one if the option `--here` is used. With this option you can precise something else, such as 'p4a-1.0.3' or 'p4a@yesterday'
- git-repository=URL: By default Par4All is cloned from the public repository. But if you have already a local clone, it may be useful to use it instead of transferring on the network or if you want to try with your own developments. For example you can use the directory name of your clone, such as `.../par4all`. For example, the advantage of this instead using `--here` is to avoid embedding parasitic files in the package.

### 6.6.3 Setup Options

- R, --rebuild: Rebuild the packages completely.
- C, --clean: Wipe out the installation directory before proceeding. Implies -R and not skipping any package.
- skip-polylib, --sp: Skip building and installing of the polylib library.
- skip-newgen, --sn: Skip building and installing of the newgen library.
- skip-linear, --sl: Skip building and installing of the linear library.
- skip-pips, --sP: Skip building and installing of PIPS.
- skip-examples, --sE: Skip installing examples.
- s PACKAGE, --skip=PACKAGE: Alias for being able to say `-s pips` (besides `--skip pips`), for example. `-sall` or `--skip all` are also available and means 'skip all', in which case only final installation stages will be performed.
- o PACKAGE, --only=PACKAGE: Build only the selected package. Overrides any other option.

**-r PACKAGE, --reconf=PACKAGE:** Always run autoreconf and configure selected packages. By default, only packages which lack a Makefile will be reconfigured. If `--rebuild` is specified, all packages will be reconfigured.

**--root=DIR:** Specify the directory for the Par4All source tree. The default is to use the source tree from which this script comes.

**-P DIR, --packages-dir=DIR, --package-dir=DIR:** Specify the packages location. By default it is `<root>/packages`.

**-p DIR, --prefix=DIR:** Specify the prefix used to configure the packages. Default is `/usr/local/par4all`.

**-b DIR, --build-dir=DIR:** Specify the build directory to be used relatively to the root directory as specify the `--root` option. Default to `build`

**--polylib-src=DIR:** Specify polylib source directory.

**--newgen-src=DIR:** Specify newgen source directory.

**--linear-src=DIR:** Specify linear source directory.

**--pips-src=DIR:** Specify PIPS source directory. When changing the directory, do not forget to reconfigure since this is when the source location is taken into account.

**--nlpmake-src=DIR:** Specify nlpmake source directory.

**-c OPTS, --configure-options=OPTS, --configure-flags=OPTS:** Specify global configure options. Default is `'--disable-static CFLAGS='-O2 -std=c99''` OR `'--disable-static CFLAGS='-ggdb -g3 -O0 -Wall -std=c99''` if `--debug` is specified.

**-g, --debug:** Set debug CFLAGS in configure options (see `--configure-options`). Please note that this option has NO EFFECT if `--configure-options` is manually set.

**--polylib-conf-options=OPTS, --polylib-conf-flags=OPTS:** Specify polylib configure opts (appended to `--configure-options`).

**--newgen-conf-options=OPTS, --newgen-conf-flags=OPTS:** Specify newgen configure options (appended to `--configure-options`).

**--linear-conf-options=OPTS, --linear-conf-flags=OPTS:** Specify linear configure options (appended to `--configure-options`).

**--pips-conf-options=OPTS, --pips-conf-flags=OPTS:** Specify PIPS configure options (appended to `--configure-options`). Defaults to `--enable-tpips --enable-pyps --enable-hpfc --enable-fortran95`. Setting this option will reset the default value. Note that several flags can be set like this : `--pips-conf-options "--enable-tpips --enable-pyps --enable-doc"`

**-m OPTS, --make-options=OPTS, --make-flags=OPTS:** Specify global make options.

**--polylib-make-options=OPTS, --polylib-make-flags=OPTS:** Specify polylib make opts (appended to `--make-options`).

**--newgen-make-options=OPTS, --newgen-make-flags=OPTS:** Specify newgen make options (appended to `--make-options`).

**--linear-make-options=OPTS, --linear-make-flags=OPTS:** Specify linear make options (appended to `--make-options`).

**--pips-make-options=OPTS, --pips-make-flags=OPTS:** Specify PIPS make options (appended to `--make-options`).

- j COUNT, --jobs=COUNT: Make packages concurrently using COUNT jobs.
- I, --no-install: Do not install any package (do not run make install for any package). NB: this might break the compilation of packages depending on the binaries of uninstalled previous packages.
- F, --no-final: Skip final installations steps in install directory (installation of various files). NB: never running the final installation step will not give you a functional Par4All build.

#### 6.6.4 Packing Options

- pack-dir=DIR: Directory where the distribution to package is currently installed. Default is to take the root of the Git repository in which this script lies.
- deb: Build a .deb package.
- T, --tgz: Create a .tar.gz archive.
- stgz: Create a source .tar.gz archive.
- arch=ARCH: Specify the package architecture manually. By default, the current machine architecture is used.
- distro=DISTRO: Specify the target distribution manually. By default, the current running Linux distribution is used.
- package-version=VERSION: Specify version for packages.
- append-date, --date: Automatically append current date & time to version string.
- publish: Publish the produced packages on the server.
- publish-only=FILE: Publish only a given file (.deb, tgz, stgz or even whatever for testing) without rebuilding it. Several files are allowed by using this option several times.
- retry-publish: Retry to publish only files (.deb and/or tgz and/or stgz) from the local directory without rebuilding them. Be sure from what is in your local directory! To be used with p4a\_pack and not with p4a\_coffee
- release: When publishing, put the packages in release directories instead of development ones.
- install-prefix=DIR: Specify the installation prefix. Default is /usr/local/par4all.
- k, --keep-temp: Do not remove temporary directories after script execution.
- pack-output-dir=DIR: Directory where package files will be put (locally). Any existing package with the same name (exact same revision) will be overwritten without prompt. Defaults to current working directory: /home/keryell/projets/WildSystems/Par4All/par4all/doc/developer\_guide

#### 6.6.5 General options

- v, --verbose: Run in verbose mode: each -v increases verbosity mode and display more information, -vvv will display most information.
- log: Enable logging in current directory.
- report=YOUR-EMAIL-ADDRESS: Send a report email to the Par4All support email address in case of error. This implies --log (it will log to a distinct file every time). The report will contain the full log for the failed command, as well as the runtime environment of the script like arguments and environment variables.

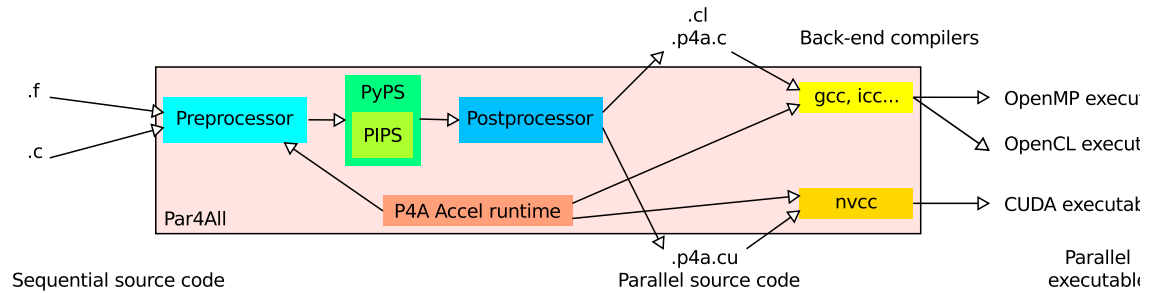


Figure 1: Intestinal view of p4a.

**--report-files:** If `--report` is specified, and if there were files specified as arguments to the script, they will be attached to the generated report email. **WARNING:** This might be a privacy/legal concern for your organization, so please check twice you are allowed and willing to do so. The Par4All team cannot be held responsible for a misuse/unintended specification of the `--report-files` option.

**--report-dont-send:** If `--report` is specified, generate an `.eml` file with the email which would have been send to the Par4All team, but do not actually send it.

**-z, --plain, --no-color, --no-fancy:** Disable coloring of terminal output and disable all fancy tickers and spinners and this kind of eye-candy things :-)

**--no-spawn:** Do not spawn a child process to run processing (this child process is normally used to post-process the PIPS output and reporting simpler error message for example).

**--execute=PYTHON-CODE:** Execute the given Python code in order to change the behaviour of this script. It is useful to extend dynamically Par4All. The execution is done at the end of the common option processing

**-V, --script-version, --version:** Display script version and exit.

## 7 p4a architecture

The global architecture of PAR4ALL is given on figure 1.

From the high-level user point of view, the sources follow this journey:

- the source files pass through the preprocessor of PIPS. C source files also pass through `p4a_recover_includes` to instrument the `#include` processing for later recovering;
- the preprocessed files pass through a splitter that creates one file per function and a compilation unit file that keeps track of all the file-global declarations;
- each function file or compilation-unit file can be parsed on demand according to the PyPS script;
- a predefined PyPS program applies many different PIPS phases on the code and regenerate the transformed sources;
- `p4a_recover_includes` is applied again as a post-processor to recover most of the `#include` work;
- the sources are postprocessed by `p4a_post_processor.py` to cope with some of the special syntax (CUDA, OPENCL...) that can not be directly represented in PIPS;
- the generated final files are copied to the final directory;

- if requested, the final target back-end compilers are called to produce the parallelized executable program.

## 8 Examples and demos

The `examples` directory contains examples and benchmarks to demonstrate the different features of PAR4ALL. Look at the local `README.txt` for more information.

There are many more examples in the validation directory of PAR4ALL (§ 9), but they are less useful for training purposes.

## 9 Validation

The validation of PAR4ALL is done at different levels.

Since PAR4ALL relies heavily on PIPS, the PIPS validation can be run into the `package/PIPS/validation` directory of the GIT working copy. Refer to the validation section of the PIPS developer guide for more information.

Note that since the validation relies on the presence of some directories, old directories that may have been automatically created in the past but are useless now may create parasitic failure entries in validation results. So it may be useful to do a `git clean -d` in `package/PIPS/validation` before running the PIPS validation. It is typically necessary also when validation directories have been moved around in the PIPSSVN and this is not well managed at the GIT level since GIT ignores empty directories.

There is a private validation in the `validation` directory of the `par4all-priv` private directory. Refer to the `par4all-priv/validation/README.txt` file to have documentation on it. It is not explained in this document for confidentiality reasons.

It is mandatory for PAR4ALL developer to run the validation before pushing big modifications to the reference GIT server.

Since PIPS is a keystone in PAR4ALL, PIPS developers should verify that their changes do not impact badly PAR4ALL by running the validation too, if possible.<sup>7</sup>

### 9.1 p4a\_validate utility

To facilitate validation, the `p4a_validate` script adds the concept of validation classes to the PIPS validation. A class is a set of validation cases with common characteristics.

TODO: For example, there could be a class for ALL the validation, for cases with the same result (e.g., `CHANGED`, `FAILED`, `PASS`, `PREVIOUS_ALL`, `PREVIOUS_CHANGED`, `PREVIOUS_FAILED`, `PREVIOUS_PASS`), etc. Once the classes are defined, set operations can be performed to combine them (e.g., unions, intersections, etc.)

TODO: Other classes can be defined directly in the validation directories with `.vclass` line-oriented regexp filter lines or with generic Python code `.vclasspy`.

`p4a_validate` has a small script interface, but the advanced user desiring more interactivity should use the Python classes directly, for example from `iPython`.

The first use of validation classes in PAR4ALL is to select from PIPS only the test cases that pass the PIPS validation. This class can be used to construct a MAT (Minimal Acceptance Test) for PAR4ALL. Other PIPS validation cases are more useful for PIPS developers, and are of less relevance to PAR4ALL validation.

#### 9.1.1 Example

Use for example with 4 processes

---

<sup>7</sup>Even if many PIPS contributors are also PAR4ALL contributors, there are PIPS contributors not familiar nor even related with PAR4ALL.

```
make validate-out -j4
```

that you can run:

- in one of the `validation` directory (from `PIPS` or in `par4all-priv`) to run all the validations cases defined in the `default` file and it generates a `SUMMARY.short` and archive it in the `SUMMARY_Archive` directory;
- in a sub-directory of the previous `validation` directory and it generates a `RESULTS` file in it.

In any case, `p4a_validate` is to be used from a `validation` directory and not from a subdirectory.

For more details on the validation process, look at the `PIPS` development guide since we use its validation infrastructure.

To display from the last validation run only the test cases marked as `changed` in `C_syntax`, use:

```
p4a_validate --file=SUMMARY_Archive/SUMMARY-last --filter='^C_syntax/' \  
--keep-status=changed --list
```

To show the test cases that changed in comparison to the reference output:

```
p4a_validate --file=SUMMARY_Archive/SUMMARY-last --filter='^C_syntax/decl' \  
--keep-status=changed --show-diff-files
```

To accept the changed validation output in `C_syntax/decl*` after a validation done from inside the `C_syntax` directory:

```
p4a_validate --file=C_syntax/RESULTS --filter='^C_syntax/decl' \  
--keep-status=changed --accept
```

When validation is complete, commit or revert to the previous state with `GIT`.

If you have run validation Occasionally, it is desirable to move all of the failed validation tests for `Control` into a new validation directory `Control-Bugs` to clean up the mainstream validation. This is accomplished with:

```
mkdir Control-Bugs  
p4a_validate --file=RESULTS/SUMMARY --list --filter=~Control/ --keep-status=failed \  
| sed -n -e 's,failed: ,,p' > files-to-move
```

Once the state `files-to-move` is complete, the database is cleaned with:

```
rm -rf Control/*.database
```

and the test cases are moved with:

```
for f in `cat files-to-move` ; do git mv $f* Control-Bugs ; done
```

### 9.1.2 Option list of `p4a_validate`

Below is a description of the usage and options for `p4a_validate`. The options are processed in the order of class construction, filtering, display and acceptance, so that actions can be piped. To track the differences between multiple validations, `pips_validate` should be run with the `-k` (history keeping) option.

None

## 9.2 p4a\_validate\_class.py validation script

The `p4a_validate_class.py` python script is a front-end to PAR4ALL validation and has several options for selecting validation tests.

Currently, available options are:

- `--pips`: validate tests which are done by default in `packages/PIPS/validation`
- `--p4a`: validate tests which are done by `par4all_validation.txt` (which exist in `src/validation`)
- `--diff`: compare the tests done with `--pips` and `--p4a` options. The list of the tests which are not done by `--p4a` options are included in `diff.txt`
- `--dir`: Validate tests which are done in `packages/PIPS/validation/directory_name`. Syntax is `./p4a_validate_class.py --dir dir1 dir2 dir3`
- `--test`: Validate tests which are given by the argument. Syntax is `./p4a_validate_class.py --test directory_test/test.f` for example
- `-h` or `--help`: Help for `p4a_validate_class.py`

Examples with `p4a_validate_class.py`:

```
python p4a_validate_class.py --pips
```

or

```
python p4a_validate_class.py --p4a
```

To use the option `--p4a`, a `par4all_validation.txt` must be previously created. This file lists all tests that the validation will do. Syntax to add a new test is: `directory_test/name_test`.

Be careful when adding a test to `par4all_validation.txt`, put the correct extension (`.c`, `.f`, `.F`, `.F90`). Don't use `.tpips`, `.tpips2...` extensions.

Example of a `par4all_validation.txt` with 2 tests:

```
Syntax/alias.f
```

```
Syntax/altret01.f
```

## 10 Branches

Since there are restrictions on the use of `/` in branch names, it is preferable to use `-` to add hierarchy.

To facilitate development and organization, there are some already defined branches:

`gcc-4.4.1` is the original GCC 4.4.1 core & Fortran in `package/pips-gfc`;

`gcc-4.4.2` is the original GCC 4.4.2 core & Fortran in `package/pips-gfc`;

`initial` is the initial commit of the PAR4ALL repository. This branch is used to set the branches for the different packages and should not be used for development.

`p4a-numerical` is a branch corresponding to a given version snapshot;

`p4a-numerical-alpha` is a branch corresponding to an alpha version of a given version snapshot;

`p4a-numerical-beta` is a branch corresponding to a beta version of a given version snapshot;

`p4a` is the branch to get the full latest PAR4ALL version with all the different components, which is the merge of the branches `p4a-own` and `p4a-packages`;

`p4a-linear` points to the import of PIPS `linear` part. Therefore, this branch should contain only files from `package/PIPS/linear` and is the merge source for the last version from `linear`;

`p4a-newgen` points to the import of the PIPS `newgen` part and should contain only files from `package/PIPS/newgen`;

`p4a-nlpmake` points to the import of PIPS `nlpmake` part and should contain only files from `package/PIPS/nlpmake`;

`p4a-own` points to the last development of the PAR4ALL files, without packages, etc;

`p4a-packages` points to the last merge of all the PAR4ALL package components;

`p4a-pips` points to the import of the PIPS `pips` part. Therefore, this branch should contain only files from `package/PIPS/pips` and is the merge source for the last version from `pips`;

`p4a-polylib` points to the import of the PIPS `polylib` part. Therefore, this branch should contain only files from `package/polylib` and is the merge source for the last version from `polylib`;

`p4a-validation` points to the import of the PIPS `validation` part and consists only of file from `package/PIPS/validation`;

`pips-gfc+gcc` points to a working PIPS-GFC implementation of the Fortran 95 extension for PIPS in `package/pips-gfc`;

`pips-gfc-4.4.1` points to the original developments of Raphaël in GCC in `package/pips-gfc`.

For tractability, there are also branches that point to specific versions of particular branches such as `p4a-0.2-alpha-nlpmake`, `p4a-own-0.1` or `p4a-packages-0.2-beta`.

## 11 Useful git tricks for Par4All

### 11.1 Files committed to the wrong branch

There are (too) many branches in PAR4ALL, right? Distinguishing the many branches of PAR4ALL is complicated; for example, it is easy to commit to the `p4a` branch instead of the correct `p4a-own` branch. The good news is that since GIT separates the commit and publish phases, one can examine the history with tools such as `gitk` to find errors before negatively impacting your colleagues.

Here is an example of a wrong `p4a` commit and how to achieve a good state before pushing to the public repository.

First create a branch to facilitate tracking:

```
git checkout -b wrong
```

Return to a clean state (use only `1 ^` to return to the previous commit):

```
git branch -f p4a p4a^
```

Move the incorrect commit to the correct branch:

```
git rebase --onto p4a-own p4a wrong
```

Then go to the target branch:

```
git checkout p4a-own
```

and merge the current work into it:

```
git merge wrong
```

and delete the helping branch:

```
git branch -d wrong
```

Magic isn't it? ☺

The wrong work is still connected to the branch `p4a`, but no longer *in* the branch. It is typical in GIT to avoid the loss of information.

## 11.2 Using git to push some branch on a test machine to try a Par4All version

Developing and testing PAR4ALL on multiple platforms at the same time may be cumbersome but hopefully GIT can help here too with the concept of distributed repositories.

For example if you develop on your local machine and want to test PAR4ALL on a test machine named `ridee.enstb.org` you can connect to, you can begin with creating some GIT clone of PAR4ALL if not already there.

Then you can create a new remote in your local GIT copy where you develop to the remote GIT, with say:

```
git remote add ridee ridee.enstb.org:some_dir/par4all
```

Pushing on a non-bare GIT repository is not a normal practice because it desynchronize the index with the working directory if the branch you push on is the checked-out one.

So you have to allow this before on the target machine with a:

```
git config receive.denyCurrentBranch warn
```

Guess you work on the branch `p4a` on `ridee`, you can begin with a clean state in the working tree with a

```
git reset --hard
```

Then you can push for example with a

```
git push ridee p4a-own
```

Then you can merge on `ridee p4a-own` into `p4a` for real test:

```
# Just in case we were not in the right branch:
```

```
git checkout p4a
git merge p4a-own
```

and compile.

You could also merge locally `p4a` and push `p4a` instead, but if you roll back `p4a-own`, you have also to roll-back `p4a`.

If you have done some roll-back in the branch locally or done some work remotely on the main branch, the history is no longer synchronized. Normally that means merging the different history. But if it is just for some throwable testing on the remote, you can remotely `git reset` the `p4a` history or use the `+` in:

```
git push ridee +p4a-own
```

that will move the `p4a-own` branch unconditionally without a merge.

Note that since GIT keep a record of the branch history it-self, you may still recover the history if it was a mistake by inspecting the output of:

```
git reflog show
```

### 11.3 The history has been rewritten on the server or how to resolve an uchrony with git

PAR4ALL is complex with many branches and sometimes some commits that should have not existed reach the development server. For example, when an error should have been fixed as in section 11.1 *before* pushing on the server...

In this case, someone change the head of the wrong branches on the server and a mail is sent to the developer list to warn everybody that the branches have been sent back in the past to discard some commits. In general the action to do, if you have not committed new stuff since this event on this branch, is something like:

```
git fetch
git branch -f p4a <some nice SHA-1 value>
```

In the general case, if you have not done some developments since the history rewriting of the p4a branch for example, you can try:

```
git fetch
git branch -f p4a remotes/origin/p4a
```

to make the local tracking branch to the server branch.

If you want to resynchronize all the branches, try:

```
# First park to a third-party unused branch to avoid interferences:
git checkout master
git fetch
p4a_git --branch-action git branch -f p4a\${suffix} remotes/origin/p4a\${suffix}
```

The following is for people willing to understand what can happen.

If you do not do that what is going to happen? If no new commit is pushed on the server and you push candidly your repository, the previously fetched wrong commits will be sent back to the server since you are in the future compared to the new reference on the server (which we sent back in the past by rewriting the history, do you remember?). Since GIT is a peer-to-peer system, your repository or the central repository are considered of the same importance and yours is trusted as so. So the wrong commits will appear back in the central repository where they will reappear for everybody... ☹ Then, everybody recurse at the beginning of section 11.3... Too bad.

If there has been some new commits on the server, the server branch is in some future compared to the point the server branch was sent back in the past but your branch (which is in the previous future ☺) seems having diverged since this point. So if you do a `git pull`, the automatic merge is likely to fail. This is why you have to reset manually the reference branch with a `git branch -f` as previously explained.

In the worst case, you have committed some work before figuring out the history has been rewritten. Then you can create a new branch to point on your work. Then you can sent back the wrong branch in the past. Then you pull from the server and you can apply back your commits by cherry-picking from the save branch or rebasing it as in section 11.1.

Have you still done something quite wrong? It is not an issue, everything is done in GIT to avoid losing work. It is somewhere. Just figure out where and how to get back on rails. ☺

To summarize this section, you need to understand that rewriting the history is globally a pain for everybody in the project. So, since a nice feature of GIT is to separate commits from pushing to a server, take some times to review commits *before* pushing them. If this is not respected, we will forbid direct commits to the central server but we will use quarantine server or branches first.

## A Various script details

In this appendix, options and other ancillary scripts are described.

## A.1 The `p4a_recover_includes` script

`p4a_recover_includes` is used to get C standard `#include` after PIPS digestion to have a smoother recompilation (for CUDA it is a requirement) and more readable code without ugly macro inclusions at the beginning of all the files.

This script can also be used to do expansion of the `#include` of PAR4ALL Accel to see the CUDA or OPENMP code that is generated.

Usage: `p4a_recover_includes [options] [<files>]`

### A.1.1 Options

- `--version`: show program's version number and exit
- `-h, --help`: show this help message and exit
- `-i, --init`: Initialize from PIPS exersizing the `#include` tables that will be used in the postprocessor
- `-E, --preprocessor`: Use this program as a preprocessor. It follows the `PIPS_CPP_FLAGS` environment variable but not `PIPS_CPP` since often the later is used to call this program. It must be the last option of this program before all the other options and parameters to be passed to CPP. This option override the `-i/--init` and `-o/--output` options. The arguments passed to the CPP are the PIPS default options, the content of the `PIPS_CPP_FLAGS` environment variable and the trailing arguments of this programm.
- `-s, --simple`: Use a simple method that should work for simple case and that avoid previous learning of the standard include files
- `-o FILE, --output=FILE`: When used in default postprocessor mode, it sets the name of the file used to output the recovered includes instead of overriding the input file

### A.1.2 Debug options

- `-v, --verbose`: Run in verbose mode
- `-q, --quiet`: Run in quiet mode [default]

## A.2 The `p4a_post_processor.py` script from Par4All Accel

`p4a_post_processor.py` transforms the PIPS output into calls to PAR4ALL Accel. It is mainly used by the `p4a` script but can be used by advanced users who need to write their own specialized PIPS scripts with `tpips` or `PyPS`.

Usage: `p4a_post_processor.py [options] <files>`

### A.2.1 Options

- `--version`: show program's version number and exit
- `-h, --help`: show this help message and exit
- `-d DEST_DIR, --dest-dir=DEST_DIR`: The destination directory name to create and to put files in. It defaults to "P4A" in the current directory>
- `-I header_list, --includes=header_list`: Specify some includes to be insetred at the begining of the file to be post processed.

### A.2.2 Debug options

`-v, --verbose`: Run in verbose mode

`-q, --quiet`: Run in quiet mode [default]

## A.3 The `optparse_help_to_tex` help documentation to $\text{\TeX}$ compiler

`optparse_help_to_tex` is a small compiler that translates a help output message of a command launched with `-h` using the `optparse` format when into  $\text{\LaTeX}$  code with section for inclusion in an article or slide generation with Beamer.

The following section should be a very demonstration of this tool indeed. ☺

Usage: `optparse_help_to_tex` [options] [<files>]

Transform the help generated by a Python program that uses `optparse` into  $\text{\TeX}$  to be included in some article or Beamer presentations :-). Have a look to `par4all/doc` Makefile for examples of heavy use and the output produced on <http://www.par4all.org/documentation> Ronan Keryell <[ronan.keryell@hpc-project.com](mailto:ronan.keryell@hpc-project.com)>

### A.3.1 Options

`-h, --help`: show this help message and exit

`--slides`: Generate the help in a format suitable for a slides presentation with  $\text{\LaTeX}$ /Beamer and proper styles.

`--article`: Generate the help in a format suitable for inclusion in a  $\text{\LaTeX}$  document.

`--no-output-usage`: By default, this script generates also the usage description. Since it may not be suitable for slide output, added this option to remove this usage so that a special manual slide can be written aside.

`--article-section-level=SECTION-LEVEL`: When generating for a  $\text{\LaTeX}$  document, this defines the starting level of the generated sections, with 0 for `\part`, 1 for `\chapter`, 2 for `\section`, 3 for `\subsection` and so on.

`-v, --verbose`: Run in verbose mode.

## B Tools, tips and tricks for developers